Available online at www.sciencedirect.com

SCIENCE DIRECT

Discrete Applied Mathematics 154 (2006) 1908–1931

DISCRETE APPLIED MATHEMATICS

www.elsevier.com/locate/dam

Online-optimization of multi-elevator transport systems with reoptimization algorithms based on set-partitioning models☆

Philipp Friese[a,1], Jörg Rambau[b,*]

[a]*Department of Optimization, Zuse Institute Berlin, Germany*
[b]*Department of Mathematics, University of Bayreuth, Germany*

Received 31 December 2004; received in revised form 18 May 2005; accepted 18 January 2006
Available online 6 June 2006

Abstract

We develop and experimentally compare policies for the control of a system of k elevators with capacity one in a transport environment with ℓ floors, an idealized version of a pallet elevator system in a large distribution center of the Herlitz PBS AG in Falkensee. Each elevator in the idealized system has an individual waiting queue of infinite capacity. On each floor, requests arrive over time in global waiting queues of infinite capacity. The goal is to find a policy that, without any knowledge about future requests, assigns an elevator to each request and a schedule to each elevator so that certain expected cost functions (e.g., the average or the maximal flow times) are minimized. We show that a reoptimization policy for minimizing average squared waiting times can be implemented to run in real-time (1 s) using dynamic column generation. Moreover, in discrete event simulations with Poisson input it outperforms other commonly used policies like multi-server variants of greedy and nearest neighbor.

Keywords: Elevator group control; Policy; Reoptimization; Online; Real-time; Simulation

1. Introduction

Our research on elevator control was motivated by the following practical application. In a large distribution center of Herlitz PBS AG, organized by the company eCom Logistik, there are two transportation modules consisting of five elevators each. These elevator systems connect the eight commissioning floors and the stacker crane terminals of the large warehouse with the ground floor. The individual elevators are controlled independently by either a FIFO or a NEAREST-NEIGHBOR heuristic (more detailed descriptions in Sections 2 and 4).

It was observed by the staff (already quite some time ago) that FIFO was problematic in high load periods whereas the NEAREST-NEIGHBOR heuristic was fast but unreliable in the following sense: surprisingly often, individual pallets were not transported at all for a very long time.

This triggered our research in how to control a single elevator so that both average and maximal flow times of pallets (the times pallets spend in the system) can be bounded. Our results in [4,21,22] suggested a policy that has

☆ Supported by the DFG Research Center MATHEON "Mathematics for key technologies" in Berlin.
* Corresponding author. Tel.: +49 921 55 7350; fax: +49 921 55 7352.
E-mail addresses: philipp.friese@zib.de (P. Friese), joerg.rambau@uni-bayreuth.de (J. Rambau).
[1] Supported by the DFG graduate program "Graduiertenkolleg 621 (MAGSI)", Berlin.

0166-218X/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.dam.2006.03.034

guaranteed maximal, and hence also average flow and waiting times, depending on the load of the system [21]. This so-called IGNORE policy bases its decisions on a current tentative schedule. Whenever a new request arrives it is buffered. Only when the current schedule is completed, a new schedule with minimal makespan is computed for all known requests. Another policy, called REPLAN revises in contrast to IGNORE the tentative schedule at the arrival of each new request. This can be shown to produce unbounded deferment of requests, and hence unbounded flow and waiting times.

Simulation results for this so-called IGNORE policy were quite satisfactory on a single-elevator system [18]. Also the results for the related REPLAN policy, though theoretically inferior, were quite good: better than IGNORE for the average flow and waiting times, worse for the maximal flow and waiting times. When, however, we put together five elevators, each controlled with the IGNORE or the REPLAN policy, the results were less encouraging. Why was this?

Before the scheduling of individual elevators can be done, each pallet has to be assigned to an elevator. This assignment, of course, greatly influences what can be achieved during scheduling. Deciding on the assignment without taking into account the consequences for scheduling led to results that more or less completely leveled off the gains of the good scheduling policy.

This led us to the conclusion that integrated assignment-and-scheduling policies were in place. In order to implement an IGNORE or a REPLAN-type integrated policy, one needs a *real-time compliant reoptimization module* for the static optimization problem for a snapshot state of the system: for all pallets already known to the system, find a tentative assignment and schedules for elevators so that some objective is minimized. The result of such an optimization, i.e., a dispatch, is then used for some time for the control decisions.

The original single-elevator policies IGNORE and REPLAN used the makespan of a dispatch as objective function, because, in contrast to the average flow time functional, there are efficient combinatorial algorithms to solve this problem for a single elevator [22].

Our first attempt in that direction was to reoptimize the makespan of a tentative dispatch in a multi-elevator system [31]. Motivated by a successful application to a warehouse single-server stacker crane in [1], we modeled the reoptimization problem as a multi-server asymmetric traveling salesman problem. We proposed a branch & cut algorithm to solve the resulting integer linear program. Computational results, however, showed that the run-times of such a reoptimization algorithm were far too large to be used as a control algorithm in real-time. Moreover, we observed in experiments that the run-times exponentially scaled with the number of pallets. That means: the more pallets the slower the algorithm, no matter how many elevators we spend.

For quite some time, we thought that exact reoptimization approaches were hopeless, and we gave up on the multi-elevator dispatching problem.

In the meantime, we successfully developed a real-time reoptimization algorithm for the online dispatching of automobile service units for the German Automobile Association (ADAC). This algorithm was able to assign 200 help requests to 80 service vehicles and to schedule these vehicles almost optimally in about 10 s. The reoptimization goal was a complicated function, put together essentially from late, drive, and overtime cost. The crucial property of the algorithm: its run-time only scaled exponentially with the number of requests per unit. The technique used was a dynamic column generation procedure, especially tuned to run in real-time [29].

Since, for the single elevator, the minimization of the average flow time or the average waiting time, or a similar function of flow or waiting times, is computationally harder than makespan minimization, we did not even think about these functionals for a multi-elevator reoptimization. (The famous discouraging result about flow time scheduling in [25] says that the weighted average flow time for jobs with release times on a single machine can only be approximated within a factor of $\Omega(\sqrt{n})$, where n is the number of requests. This contrasts the polynomial solvability of the single-elevator scheduling problem, as shown in [22]).

After the success in the ADAC project, however, it occurred to us that the multi-elevator reoptimization could essentially be done with a modified algorithm from the ADAC project, thereby reoptimizing some function on the flow or waiting times. First results are documented in [14]. (Interestingly enough, a makespan minimization turned out to be conceptionally harder in the dynamic column generation model, with a selection variable for each possible tour of a server through a set of requests, than an average waiting time minimization).

In this paper, we finally present the results of extensive simulation studies on how these new reoptimization policies that use an adopted reoptimization algorithm form the ADAC project, perform w.r.t. to the considerably harder real-time requirement of 1 s answer time. Our simulation studies have been carried out w.r.t. an idealized, symmetric elevator system, removing the capacity restrictions in the waiting queues. This was to emphasize the genuine role of the interplay

between assignment and scheduling, untweaked by specialities of the real system that might not be transferable to other systems.

As input distribution we used Poisson arrivals. It turned out that the performance of reoptimization policies, in particular w.r.t. the maximal waiting times, heavily depends on the reoptimization objective. The best results over all could be achieved by minimizing the sum of squared waiting times in each reoptimization. Since the reoptimization algorithm is flexible in that respect, more complicated objectives for the reoptimization models are conceivable. We investigated some of them, but since the most significant improvements stem from the minimization of the sums of squared waiting times, we stick to the "pure" objectives in this presentation.

Currently, we aim at applying the key learnings of this work to the real elevator system in the distribution center of Herlitz.

1.1. Related work

Optimization literature on elevator control in particular is surprisingly sparse. In [11,12], elevators with higher capacities for human passengers are controlled with reinforcement learning techniques. Their setting is different from ours in that the neither the number of passengers on a floor nor their destinations are known to the controller.

Only after the development of so-called destination control, telling the controller how many people want to go to which floor (at Schindler Lifts Ltd., productive in 1996), it became more interesting to compute optimal tentative schedules. This was, e.g., done in [26]: a reoptimization technique based on branch & bound was used to generate tentative stop sequences for a single elevator minimizing the average waiting time of the passengers. In the same paper and in [27], a concept was introduced how this reoptimization can be integrated into an AI-based formal online control system. Still, the control of elevators with capacities larger than one poses completely different optimization problems than the control of a unit capacity elevator.

Another model that fits the elevator control problem is the general online dial-a-ride problem, which is usually investigated with competitive analysis. Several results for a single elevator are available: competitive algorithms are only known if the goal is to minimize the long-term makespan [4] or the weighted sum of completion times (measured from a global time zero) [28]. For minimization of the maximal or average flow times, it is known that the aforementioned IGNORE policy is guaranteed to perform well under reasonable load [21], but no competitive online algorithm can exist. That the makespan reoptimization can be done in polynomial time, at least if stopping and starting times of the elevator are neglected, was shown in [22]. Even with additional starting and stopping times, the same algorithm yields asymptotically almost surely an optimal solution [10].

There is a vast literature on dynamic vehicle routing problems. We can view elevators as vehicles traveling through a set of requests, where the distances are asymmetric, given by the connecting moves. This model was used in the online control of the (two-dimensional) movements of an automated stacker crane, serving a warehouse (see [1]). The evaluation of the resulting reoptimization policy was done by simulation experiments: significant improvements over a priority-based control policy could be achieved.

Other dynamic vehicle routing problems have been used as test cases for models that are supposed to deal appropriately with the dynamics of the system. An overview of what principle difficulties have to be taken into account can already be found in [33].

There are several heuristic approaches that are all some kind of approximation of reoptimization policies. Tabu-search-based methods are proposed in [15]. Diversion issues (diversion = changing the next customer of a vehicle) are investigated in [23]. An overview over some heuristic methods can be found in [16]. Another local search method is used as the reoptimization module in [36], but the main focus in that paper is how the concurrency of certain events is resolved in the controller.

A newer type of dynamic heuristic is the multiple plan approach presented in [5]. This heuristic might be especially interesting if the system might run into an infeasible state, e.g., when hard time windows are present.

Another attempt to avoid too complicated reoptimization by feedback mechanisms in logistic queueing networks was proposed in [32] for the management of a fleet of vehicles to service a known set of loads. This approach might help whenever exact reoptimization techniques cannot be implemented to run in real-time.

To theoretically evaluate the expected performance of control policies w.r.t. some input distribution, Markov decision models are certainly theoretically appropriate. Alas, the computational effort for an exact evaluation of the expected performance of a policy is for all but the most trivial vehicle routing problems prohibitive. Nevertheless, based on a

Markov decision model, in [30] a decomposition heuristic is designed that can be used for moderately sized vehicle dispatching problems for the delivery of goods. This heuristic, however, is only evaluated in the case when the decisions irrevocably fix the complete routes for the vehicles until the routes have been traversed completely. In a sense, the considerations in that paper have been restricted to IGNORE-type policies.

Theoretical results on the basic dynamic traveling repairman problem w.r.t. the minimization of the expected total waiting time can be found in [8,9]. Here, the input distribution of requests is given by Poisson arrivals in each point of a bounded area in the Euclidean plane. In our language, a variant of FIFO is stable and optimal as the load approaches zero, whereas variants of IGNORE are stable and provably good in heavy load.

In spite of some successes in the theoretical evaluation of policies for vehicle routing, for large-scale systems simulation experiments seem still unavoidable. A data model that is suitable for the simulation of an elevator system must take into account that elevators are moving objects and processing severs at the same time. A clean simulation data model for this can be found in [35]. Nevertheless, it would be desirable to find at least computational bounds on the expected performance of a policy. Maybe the method in [7] could produce a suitable tool, but so far nothing in this direction is in sight for elevator control.

We close this section with a reference that is in sync with our aim to design proper reoptimization policies for dynamic dispatching problems like the multi-elevator control problem. The main encouragement for the use of exact reoptimization policies can be found in [6]. Based on a thorough analysis of various aspects of dynamic and stochastic vehicle routing, the authors come to the following conclusion: "A reoccurring finding in the analysis is that static vehicle routing methods when properly adapted can yield optimal or near optimal policies for dynamic routing problems". In this spirit, we try to find out the right models for reoptimization so that the corresponding reoptimization policies perform well.

1.2. Our contribution

We present simulation experiments that compare the performance of various policies for the dynamic elevator dispatching problem (defined below) that have not been thoroughly compared before.

Some of the policies are common in practice, some are new. For the first time, we quantitatively show that exact reoptimization, where assignment decisions and scheduling decisions are integrated into one model, firstly, improves the performance of a multi-elevator system significantly and can, secondly, be implemented to run in real-time.

More specifically: exact reoptimization w.r.t. the minimization of the sum of squared waiting times leads to a stable system for very high loads, when other heuristic policies like variants of NEAREST-NEIGHBOR let the system overflow already. Thereby, the reoptimization of the sum of squared waiting times leads to the best maximal individual waiting times (fairness). We furthermore found out that, under high load, heuristic reoptimization by means of best-insertion or two-exchange methods cannot completely cope with exact reoptimization.

Our exact reoptimization can be implemented so that all reoptimizations can finish with an optimality gap below 5% in only 1 s whenever the number of requests per elevator is not larger than five. Our implementation is a column generation algorithm with dynamic pricing control, based on a vehicle dispatching algorithm ZIBDIP, used for the real-time dispatching of automobile service units.

Although most people will not be surprised that a first-in-first-out (or first-come-first-serve) policy is inefficient, we show quantitatively how disastrous the use of such a policy is. In our idealized elevator system with 16 floors the following can be observed: in order to achieve a fixed average waiting time, without any optimization one needs around 60% more elevators than with rigorous mathematical reoptimization; with very common, heuristically optimizing policies one needs around 20% more elevators.

1.3. Structure of the paper

In Section 2, we describe the idealized elevator system under consideration and the aspects that have been simplified in contrast to the motivating real system. In Section 3, we present models for the underlying dynamic and the static multi-elevator dispatching problem, which yields mathematical formulations of both our special control problem and the corresponding reoptimization problems. Section 4 introduces and classifies some policies. The simulation environment as well as the computational results are provided in Section 5. We add some remarks about possible extensions in Section 6, before we close the paper with our conclusions in Section 7.

2. The system of study

The multi-elevator system (sometimes called an "elevator group") under consideration is an idealized system. It consists of k elevators traveling at unit speed. Stopping, entering, exiting, starting, and turning may take individual additional time, but we will not mention this in the sequel, for the ease of exposition. Each elevator has capacity one, i.e., it can only carry one request at a time. There is a global waiting queue for each floor with infinite capacity. Requests become known to the system control when they arrive in this queue. In front of each elevator, a separate local waiting queue with infinite capacity exists. From the global waiting queue, each local queue, and hence each elevator, can be reached in the same constant amount of time. From the local waiting queue in front of an individual elevator, only that elevator can be reached. In all waiting queues, requests cannot overtake each other.

In the motivating application, each elevator's local waiting queue has capacity one, the global waiting queue is a circular conveyor moving the pallets around the elevator system, passing the elevators' local waiting queues one by one. Therefore, in reality the time to reach the local waiting queue of an elevator depends on the elevator.

The reason for studying the idealized system in detail is that we would like to understand better the core of the underlying more general online dispatching problem, unmixed with the very special situation in the motivating application that might not be present in similar applications. A thorough investigation of the real system is currently work in progress, exploiting principles revealed in this paper.

3. Modeling

We present models for the dynamic and the static multi-elevator dispatching problem.

3.1. Model for the dynamic multi-elevator dispatching problem

An instance of the dynamic multi-elevator dispatching problem consists of a multi-elevator system as described above and a distribution of requests.

Requests arrive over time, and no information is available about future requests. Requests r are triples $r = (\tau(r), o(r), d(r))$ with enter time $\tau(r)$, origin floor $o(r)$, and destination floor $d(r)$. The enter time $\tau(r)$ of r is the time at which r becomes known and is available in the global waiting queue on its origin floor $o(r)$. The transportation task is to carry r from floor $o(r)$ to floor $d(r)$ without pre-emption, i.e., requests can only be dropped at their final destination.

An online algorithm as well as a dynamic control policy for such an elevator system must make the following two control decisions over and over again:

(1) Whenever a new request arrives, choose an elevator to serve this request (online assignment).
(2) Whenever an elevator is idle, either choose a yet unserved request in one of the elevator's local waiting queues to be served next or decide that the elevator be idle for a specified amount of time (online scheduling; see Fig. 1).

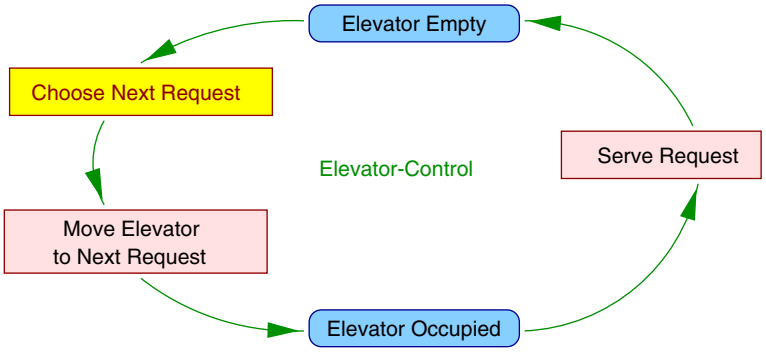

Fig. 1. The basic control cycle for the control of a single elevator.

The difference in the literature between an online algorithm and a (control) policy is that an online algorithm has never any information about the input distribution, whereas a policy may know the input distribution. Moreover, online algorithms are usually evaluated by competitive analysis, which measures their worst-case deviation from an a posteriori offline-optimum solution on finite input sequences, whereas policies are usually evaluated by expected performance w.r.t. to an input distribution, an average measure. We will stick to the term policy, because our experimental evaluations are average performances w.r.t. an input distribution, although all policies considered in this paper are in fact online algorithms, i.e., they do not use any knowledge about the input distribution.

A *server move* m is a quadruple $m = (\theta(m), o(m), d(m), r(m))$, where $\theta(m)$ is the start time of the move, $o(m)$ is the origin floor of the move, $d(m)$ is the destination of the move, and $r(m)$ is either the carried request of the move or EMPTY if m is an empty move. A *transportation protocol* $P = (P^s)_{s\in S}$ is a vector that consists of k sequences $P^s = m_1^s, m_2^s, \ldots, m_{n^s}^s$ of server moves, one sequence for each server $s \in S$. These moves must be consecutive and chronological, i.e., $o(m_{j+1}^s) = d(m_j^s)$ and $\theta(m_{j+1}^s) \geqslant \theta(m_j^s) + |d(m_j) - o(m_j)|$ for all $1 \leqslant j < n^s$ and all $s \in S$. Moreover, since no pre-emption is allowed, $o(m) = o(r(m))$ and $d(m) = d(r(m))$ whenever $r(m) \neq$ EMPTY. We denote by $m_P(r)$ the move in P that carries request r. For every finite realization of the request distribution, each policy induces a finite transportation protocol transcribing the effect of the policy's decisions.

We consider the following cost functions on transportation protocols:

1. The average waiting time of a transportation protocol P for a set R of n requests is given by
$$c^{\text{avgwait}}(P) := \frac{1}{n} \sum_{r\in R} (\theta(m_P(r)) - \tau(r)).$$
2. The maximum waiting time of a transportation protocol P for a set R of n requests is given by
$$c^{\text{maxwait}}(P) := \max_{r\in R} \{\theta(m_P(r)) - \tau(r)\}.$$
3. The average squared waiting time of a transportation protocol P for a set R of n requests is given by
$$c^{L_2\text{-wait}}(P) := \sqrt{\frac{1}{n} \sum_{r\in R} (\theta(m_P(r)) - \tau(r))^2}.$$

Essentially, all three objectives are L_p-norms of the waiting time vector, whose entries are the individual waiting times of the requests in R. The average waiting time is the nth fraction of the L_1-norm, the maximal waiting time is the L_∞-norm, and the L_2-waiting time is the $\sqrt{n}$th fraction of the L_2-norm of the waiting time vector.

We remark that, even for a single server, for none of these objectives there are competitive online algorithms. This is because, no matter where an online algorithm has positioned its server, a malicious adversary could issue a request with large enter time somewhere else; since the enter time is large, an offline optimum could serve that request with waiting time zero by moving the server there in advance, whereas the online algorithm still has to move its server to the request, incurring a strictly positive waiting time.

We further note, that an optimal policy for our dynamic dispatching problem that minimizes the expectation of one of the average cost functions—let alone the maximal waiting cost—is not known. Classical computational methods like value or policy iteration seem prohibitive since the number of states necessary to describe the system is astronomic.

We resort to an evaluation of special policies by discrete event-based simulation in order to estimate the expected performance of the policies under consideration.

3.2. Model for the static multi-elevator dispatching problem

In this section, we present an integer linear programming (ILP) model for the underlying offline optimization problem starting in an arbitrary system state. An instance of this static multi-elevator dispatching problem is a system state of the dynamic problem. Such a state consists of

1. A set R of requests that are not yet being carried by a server.
2. A subset $R' \subseteq R$ of requests that are already irrevocably assigned to a server, i.e., that are already in one of the servers' local waiting queues. For $s \in S$, the set $R'_s \subseteq R'$ is the set of requests already assigned to server s.

3. For each server s, the request $r_0(s)$ that it is currently carrying.
4. For each server s, the time $\theta_0(s)$ and position $o_0(s)$ at which it will be empty next. If the server is empty then this is the time and position at which it can stop next (usually the next floor in its current direction).

What is such a model good for? Since we currently have no clever idea how to make use of the input distribution (and we would not trust that information in practice anyway), we plan to base our decisions in the dynamic problem on what we have in hands: the current system state. Using this as an input, we will solve an optimization problem w.r.t. some objective in order to produce a currently optimal dispatch, i.e., a transportation plan for the future, from which we can draw all decisions until the system state changes (i.e., at the arrival of a new request).

The model is based on set partitioning w.r.t. the set of all requests in R into *server tours* for the servers in S. That is, we have a variable x_T for every *feasible server tour*. This allows us to include *precedence constraints* and *preassignment constraints* into a feasibility check for tours that are *implicitly enumerated* rather than modeled mathematically. We explain this in more detail in the following.

A *server tour*, or *tour* for short, is given by a server s and a sequence T of requests with the meaning that server s serves the requests of the sequence in the order of the sequence. Without loss of generality, for the objectives under consideration (norms of waiting time vectors) each server traverses such a tour in the fastest possible way, since there are no conflicts between various servers, and all requests have been released already. That is, all server moves in a tour including their start times are w.l.o.g. given by the sequence of requests to be served. As start times of the moves we will always assume the earliest possible start time.

The start time of a tour of server s is the time $\theta_0(s)$ at which the server will have finished its service on the request $r_0(s)$, i.e., the request currently carried by s. All start times of the following moves in a tour are uniquely given by the distances (in floors) the server has to travel in order to complete the moves. The *cost* c_T of a tour T can be any cost function. We mainly consider the sum of waiting times and the sum of squared waiting times of the requests in T.

A server tour is *feasible* if the *precedence constraints* of the global waiting queues and the *preassignment constraints* for the requests in $R' \subseteq R$ are satisfied. The precedences stem from the motivating application because pallets cannot overtake each other on a conveyor belt. The precedences enforce that all requests with the same origin floor must appear in the tour in the order of their enter times in the global waiting queue on their common floor. The preassignment constraints are induced by requests that have already entered the local waiting queues of the elevators and can therefore not be assigned to a different elevator anymore. If, e.g., a request r is in R'_s for a server s, then a tour for s is only feasible if it contains r, and the tour for the other servers are only feasible if they do not contain r.

A *feasible dispatch* D is a set of feasible tours, one for each server, so that each request is in exactly one tour. The cost of a feasible dispatch is the sum of the costs of its tours.

We call $\mathcal{T}$ the set of all feasible tours (including void tours in which the set of served requests is empty), and $\mathcal{T}_s$ is the set of feasible tours for server s for $s \in S$. Let $A = (a_{rT})_{r\in R, T\in\mathcal{T}}$ be the incidence matrix of requests and tours and let $B = (b_{sT})_{s\in S, T\in\mathcal{T}}$ be the incidence matrix of servers and tours.

Let now $x \in \{0,1\}^{\mathcal{T}}$ be a decision variable with the meaning $x_T = 0$ if and only if T is in a dispatch. Moreover, let $c \in \{0,1\}^{\mathcal{T}}$ be the vector of the corresponding tour costs. With this notation, we can formulate the static multi-elevator dispatching problem as the following integer linear program:

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax = 1 \quad \text{(partitioning of requests)} \\ & Bx = 1 \quad \text{(partitioning of servers)} \\ & x \in \{0,1\}^{\mathcal{T}}. \end{aligned}$$

In the case when c_T is the sum of waiting times or the sum of squared waiting times in T, the cost $c^T x$ of a feasible solution x is the sum of waiting times or the sum of squared waiting times over all requests. The former objective is equivalent to the average waiting time, and the latter objective is equivalent to the L_2-waiting time, introduced for the evaluation of transport protocols in Section 3.1.

Of course, the dimension of this program is huge. The variables, however, can be generated implicitly. This so-called dynamic column generation procedure is by now a well-established technique. The two big advantages of a model

based on tour variables, rather than on multi-commodity flow variables, are, first, the possibility to model difficult, non-linear side constraints and costs, and, second, the smaller integrality gap between the model and its LP relaxation.

4. Policies for the dynamic elevator dispatching problem

In this section, we introduce policies for the dynamic elevator dispatching problem. Some of them are commonly used in practice, some appear in the literature. We additionally propose reoptimization policies that use exact methods for reoptimization; such policies are very often discarded based on the assumption that the corresponding solution algorithms cannot be implemented to meet real-time requirements like 1 s answer times. That exact reoptimization methods can indeed be real-time compliant in certain applications was proven for the dynamic dispatching of automobile service units for the ADAC in [29,19]. Actually, a more serious problem of such policies is their sometimes undesired dynamic behavior, as was shown in [21], which heavily depends on the specific problem, as we will see in the computational results in Section 5.3.

According to the two basic control decisions a policy has to make, the problem can be hierarchically decomposed: first for every new request we need an assignment decision, second each server must make scheduling decisions for each request assigned to it. It is in principle possible, to put together a policy by combining an arbitrary assignment policy with an arbitrary scheduling policy. This decomposition is often called *cluster-first-schedule-second*.

Of course, the assignment decisions influence what will be possible in the scheduling decisions of the individual servers. Therefore, the assignment decisions and scheduling decisions might also be made in a coordinated way. We call policies of this type *integrated policies*.

4.1. Assignment policies

We list some natural assignment policies that are commonly used. The possible assignment decisions are few, so the core of the problem is not in the computational effort in some optimization problem but is completely in the unknown future.

4.1.1. Rule-based policies

This is the easiest assignment policy: just assign to the ith request server with number $i \bmod k$. This policy does not base its decisions on any calculation related to efficiency. Therefore, it is believed that the expected result is very inefficient. Nevertheless, this policy ROUND-ROBIN is widely used.

4.1.2. Greedy policies

In this policy, for each server the number of yet unserved requests is continuously updated (or some other load measure: shortest queue on the origin floor, least amount of loaded floors to be traveled for the already assigned requests, etc.). Assign a new request to the—in that sense—least loaded server. In our experiments, we use in BALANCE the server with the smallest total travel time to the request if it is appended to its current tour.

4.2. Scheduling policies

Scheduling policies can be drawn from the literature on the online dial-a-ride problem [13,4,20]. Most investigations, however, concern competitive analysis for online algorithms. Simulation studies can be found in [18]. Policies for scheduling the individual servers get the assignment policies' outputs as their inputs. This makes the assignment decisions so important.

4.2.1. Rule-based policies

This class of policies makes its control decisions on the basis of external rules with no calculations according to cost. The main example of such a policy is FIFO: all requests are served in the order of appearance.

In the Operations Research community it is quite accepted that such a policy leads often to inefficient system behavior. It is, however, sometimes claimed that it implements more "fairness" in the system than other policies heading for efficiency.

It will be shown in Section 5.3 that if at all their is fairness to FIFO it is fairness on a very low level of service quality: all requests have to wait long and the system's throughput (the rate of requests that the system can handle without overflowing) is very low.

Other rule-based policies include PRIORITY: each request belongs to a certain priority class, and requests with higher priority are served first, where ties are broken according to FIFO. Our experience is that for PRIORITY it is often overseen that it is exactly the tie-breaker FIFO that leads to inefficient behavior of PRIORITY [1,3].

4.2.2. Myopic policies

This class of policies tries to find a control decision that incurs the least immediate cost. The most famous one is NEAREST-NEIGHBOR: always serve the closest request next. That means, NEAREST-NEIGHBOR minimizes the length of the server's empty move to the next request. For unit speed, this is the same as minimizing the start time of the next loaded move.

The NEAREST-NEIGHBOR policy is usually observed to have very small average waiting times, but individual waiting times can become quite large. Instances can be constructed in which some requests are never served.

4.2.3. Heuristic reoptimization policies

Here, and in the next section, the decisions are based on the solution of a reoptimization problem that is an instance of a static single-elevator control problem w.r.t. some fixed cost function and maybe additional side constraints. Sometimes the cost function and the side constraints are exactly as in the dynamic problem. We call this *congruent reoptimization*. One cost function in the literature used for reoptimization that is not congruent to the dynamic problem is the makespan [21]. The reoptimization problem is constructed at the arrival of each new request. The policies in this section solve this problem by heuristic means, i.e., they do not give any information about the optimality gap of the solution obtained.

The most prominent representatives are probably REOPT-BESTINSERT and REOPT-TWOOPT. At the arrival of a new request, the policy REOPT-BESTINSERT starts with the previously computed schedule. It removes all requests that have been served already and inserts the new request so that the resulting schedule has the smallest possible cost. The policy REOPT-TWOOPT goes further: it iteratively exchanges the position of two requests in the resulting schedule so that the schedule stays feasible and that the cost decreases; this is done until no progress can be made this way anymore.

It has been stated in the literature [36, p. 5] that heuristic reoptimization might help to leave some slack in the solutions that lead to more robustness w.r.t. to possible future requests. We will see that this hope is not always based on facts.

4.2.4. Exact reoptimization policies

As in the previous section, decisions are based on the outcome of a reoptimization procedure, but here we use exact methods in the sense that the optimality gap can be computed.

For dial-a-ride problems like the static single-elevator dispatching problem, there are ideal cases where the reoptimization problem w.r.t. makespan minimization can be solved in polynomial time [22]. For the waiting time-based objectives the reoptimization problems are mostly NP-hard, e.g., because the traveling repairman problem can be reduced to them.

A general solution method is branch & bound. The performance heavily depends on the specific problem. We say that an implementation of such a reoptimization procedure is real-time compliant if it produces a gap below 1–5% within the required answer time, which is 1 s in our case.

4.3. Integrated policies

In this section, we introduce some integrated policies that include into their assignment decisions the impact on the scheduling problem for the individual elevators. To this end, they maintain a current schedule for each elevator (i.e., a tentative dispatch) that is updated immediately after each assignment decision. This assignment decision takes the resulting updated schedules into account beforehand.

4.3.1. Greedy-type policies

Greedy-type policies are characterized by the fact that they never change the start time of a request in an elevator's schedule: new requests are either inserted so that the following requests are not postponed or they are appended. As examples, we introduce two greedy-type policies GREEDY-LATE and GREEDY-DRIVE. Both assign a request as early as possible into a gap in an elevator's schedule. If this cannot be done, both append the new request to one of the schedules. The policy GREEDY-LATE picks the elevator such that the new request has the smallest possible waiting time. The policy GREEDY-DRIVE chooses the elevator so that the length of the empty move to the new requests is as small as possible. Ties are broken in both cases in a round-robin fashion. While the former policy is quite natural by itself, the latter policy is motivated by the common opinion that avoiding empty moves leads to a more efficient system.

4.3.2. Heuristic reoptimization policies

Heuristic as well as exact reoptimization policies are allowed to change the current dispatch according to all available degrees of freedom. This means that in order to evaluate an assignment decision, all previously made tentative scheduling decisions, if not yet implemented irrevocably, can be revised.

More formally, at the arrival at each new request a static multi-elevator dispatching problem is constructed. The input for this offline optimization problem is the current state of the system in the sense of Section 3.2. As in the single-server case, a feasible solution can be obtained by taking the current dispatch, removing all served requests, and inserting the new request at least possible cost. This leads to the REOPT-BESTINSERT policy. Similarly, there is a REOPT-TWOOPT policy that tries to iteratively reduce the cost of a dispatch by exchanging the positions of two requests.

At times it is claimed that in practice very often REOPT-TWOOPT yields results very close to what an exact optimization can do.

4.3.3. Exact reoptimization policies

Here, we reoptimize so that provable optimality gaps for the solutions to the reoptimization problems are available. Again we say that an implementation of a reoptimization procedure is real-time compliant if it produces a gap below 1–5% within the required answer time, which is 1 s in our case. Our main tool for obtaining such tight quality guarantees is dynamic column generation for an LP relaxation of the ILP model in Section 3.2. The input is again given by the current state of the system.

Note that when the cost of a tour is the sum of waiting times, an optimal dispatch also minimizes the average waiting time over all requests. That means, for minimizing the average waiting time in the dynamic problem such a tour cost defines a congruent reoptimization. The same is true for the L_2-waiting time in the dynamic problem when the cost of a tour is chosen to be the sum of squared waiting times.

It is not true that congruent reoptimization is always good in expectation. We will see, however, that for some objectives in the dynamic problem it is the best that we currently know.

5. Simulation experiments

In this section, we compare a selection of policies described in the previous section. Before presenting the figures, we describe the implementations and the computational environment.

5.1. The implementation of exact reoptimization

For a very long time it was claimed that an ILP algorithm for a complicated offline optimization problem like the static multi-elevator dispatching problem could not be implemented to meet real-time requirements like 1 s answer time. In [29], however, a large-scale real-world application was presented, in which an ILP algorithm based on dynamic column generation could solve static service vehicle dispatching problems with 80 vehicles and 200 service requests in 10 s to (near) optimality.

The special algorithm ZIBDIP in that work used a concept called *Dynamic Pricing Control* that, for that particular application, led to a rapidly converging column generation procedure. It occurred to us that many characteristic aspects also appear in the static multi-elevator dispatching problem. The implementation used for the exact reoptimization policies REOPT-ZIBDIP, considered in this paper w.r.t. c^{avgwait} and $c^{L_2\text{-wait}}$, resp., are therefore based on ZIBDIP.

Table 1
Investigated policies and their shortcuts

Shortcut	Policy
A	ROUND-ROBIN and FIFO
B	BALANCE and NEAREST-NEIGHBOR
C	BALANCE and REPLAN
D	BALANCE and IGNORE
E	GREEDY-DRIVE
F	GREEDY-LATE
G	REOPT-BESTINSERT w.r.t. c^{avgwait}
H	REOPT-TWOOPT w.r.t. c^{avgwait}
I	REOPT-ZIBDIP w.r.t. c^{avgwait}
J	REOPT-BESTINSERT w.r.t. $c^{L_2\text{-wait}}$
K	REOPT-TWOOPT w.r.t. $c^{L_2\text{-wait}}$
L	REOPT-ZIBDIP w.r.t. $c^{L_2\text{-wait}}$
M	REOPT-ZIBDIP w.r.t. c^{avgwait} and Extreme Load Model
N	REOPT-ZIBDIP w.r.t. $c^{L_2\text{-wait}}$ and Extreme Load Model

Precedence constraints and preassignment constraints had to be added to the column generation procedure. The implementation of our reoptimization algorithm is real-time compliant whenever the number of unserved requests per elevator is around five. In order to guarantee real-time compliance of the policy, the reoptimization run is interrupted after 1 s. If the reoptimization is interrupted this way prematurely, then the algorithm may not have closed the optimality gap below 5%. In these cases, it represents only a heuristic reoptimization method. However, in terms of the quality of the reoptimization results, REOPT-ZIBDIP dominates both REOPT-BESTINSERT and REOPT-TWOOPT, since both heuristics are used in ZIBDIP's preprocessing anyway.

5.2. Computational environment

We simulated cargo elevator systems with five elevators operating on 16 floors in 12 samples with 2 h of simulation time each. Thereby, we investigated low system load as well as high system load. The requests were generated randomly with exponentially distributed inter-arrival times. For the results, we monitored the *average waiting time*, the *average squared waiting time*, the *maximum waiting time*, and the *load histogram* (how many known requests are unserved at a point in time) of each elevator system. All waiting times can be seen as a measure for the Quality of Service (QoS), as the load histogram indicates the stability of the system, since in a real system the waiting queues have finite capacities.

We call a policy *stable* if it is able to reduce the number of known unserved requests below a constant number again, no matter how long the system has been running already. For an unstable policy, the "downpeaks" of the system load over time are constantly increasing. In the histograms, this can be easily checked by looking at the lower envelope of the graph: if the lower envelope of the graph is bounded by a constant, then the policy is (experimentally) stable, otherwise not.

In order to be able to display the results of the various policies side by side, we use shortcuts for the policies (see Table 1).

We performed three consecutive simulation series. In the first series, we compared the policies A–D as representatives for hierarchical policies, making assignment decisions and scheduling decisions independently. In the second series, we compared the "winner" of this series with the policies E and F, representing heuristic ideas to integrate assignment and scheduling decisions. Finally, in the third series, we compared the "winner" of the previous series with the policies G–N, representing the integrated reoptimization policies. Note that the load histogram of each policy is only displayed once in the results.

5.2.1. Hardware and software setup

All simulations have been carried out on a 2.4 GHz Intel Xeon machine, equipped with 2.0 GB of RAM, running Linux as operating system (kernel version 2.6.8). The source codes were compiled using the GNU C++-compiler

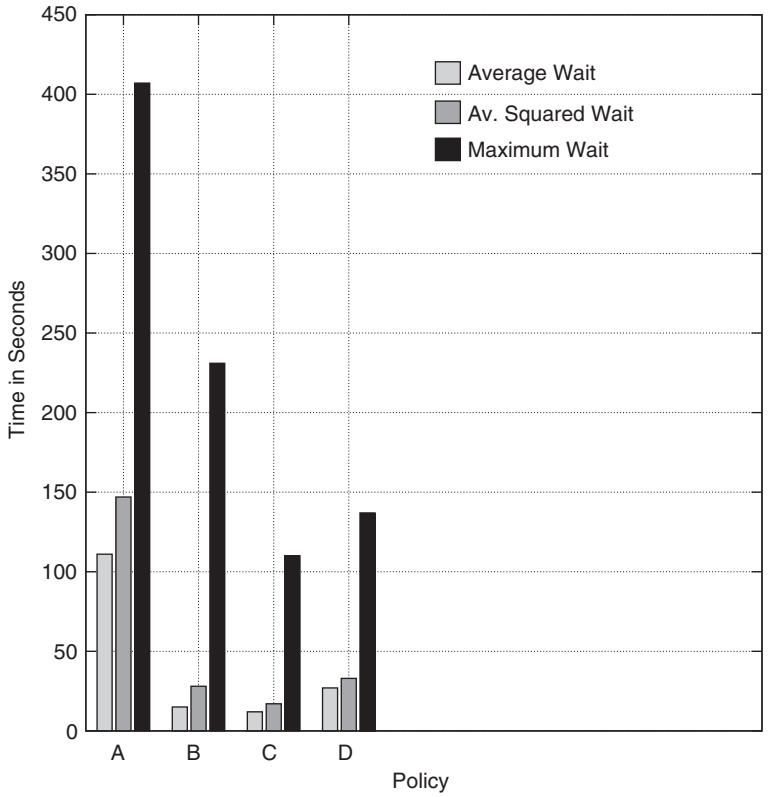

Fig. 2. Results of policies for low system load.

gcc, version 3.3.4 (pre 3.3.5 20040809) [17] with the compiler flag -O2. The algorithm ZIBDIP used the LP and ILP solver CPLEX version 9.0 [24]. We disabled the presolving mechanism of CPLEX by setting the parameter CPX_PARAM_PREIND to CPX_OFF, because we had some trouble with extracting correct dual prices when the presolving mechanism was active.

5.2.2. Simulation environment

Our simulation tool is based on the "low level" simulation library AMSEL, a callable C-library to design event-based simulation programs. The input data of a simulation experiment consists of a set of *event points*, a set of *modules*, and a collection of *requests*. Each request becomes an *object* which flows through the system via the event points. For each event point a method is specified that derives for each object entering that event point a successor event from the current state of the system. If an object is logically located on an event point (i.e., the event "happens" for this object), then the event is stored in the chronologically sorted global *event list* together with a time stamp and the object identifier; the object stores the same event as its current event.

The basic flow of objects is modeled as follows: the currently next event in time is read from the event list. Then the successor event is derived together with the point in time when this event should be processed. Now, the object updates its current event to the successor event, and the successor event is inserted into the global event list; the old event is deleted from the list. Modules are closed regions in the system where the number of objects inside is constrained by a capacity value. Modules are entered through entry event points and left through exit event points. In order to compare different policies for elevator systems, it is possible to simulate multiple system configurations simultaneously. For more details on AMSEL see [2]. For more background on the underlying principles of discrete event simulation, see [34].

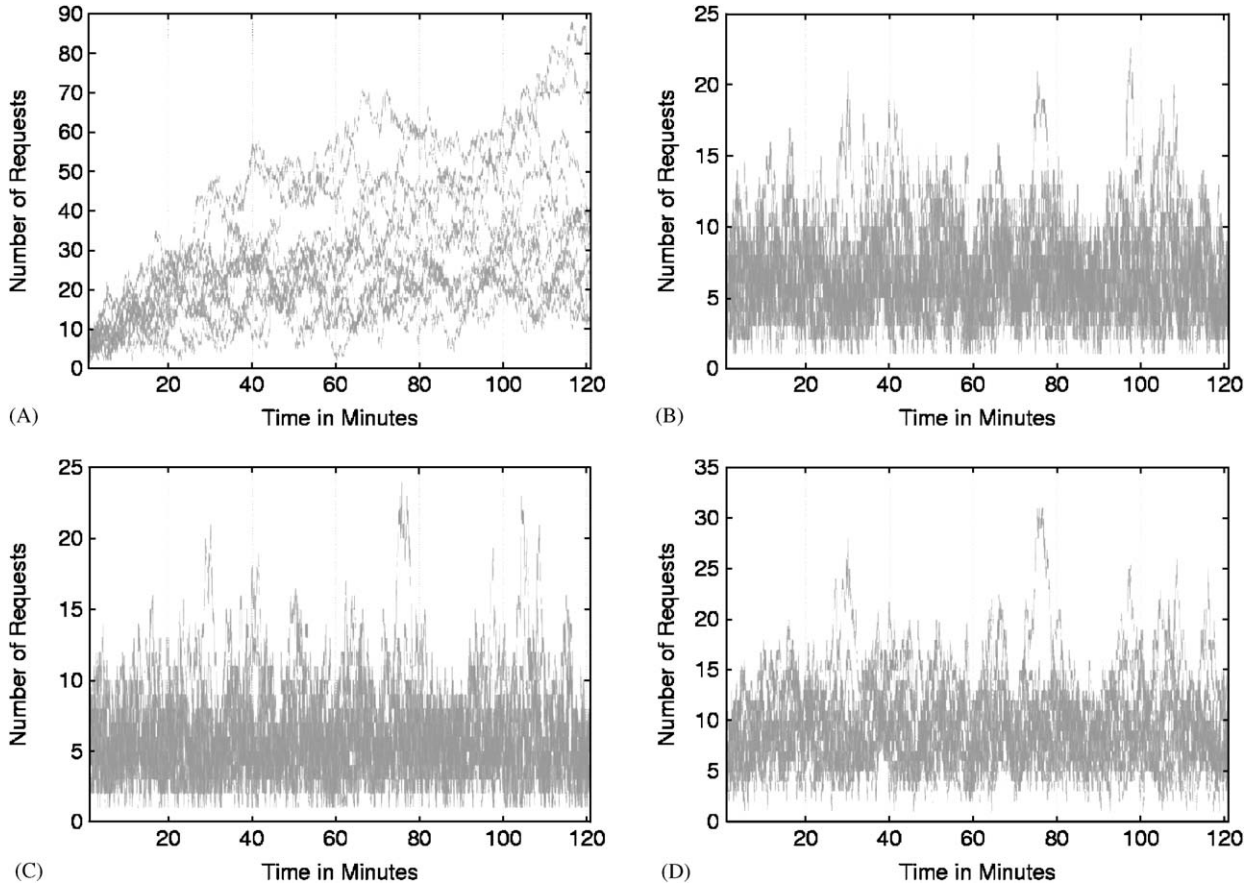

Fig. 3. Load histograms of policies at low system load: (A) policy A, (B) policy B, (C) policy C and (D) policy D.

5.3. Computational results

In this section we present our computational results of the simulations. We start with the evaluation of policies for low system load. Afterwards we do the same for high system load.

Finally, we evaluate in an extra simulation series, how much capacity (in terms of elevators) at a given QoS (in terms of average flow time) can be saved by using exact integrated reoptimization REOPT-ZIBDIP w.r.t. $c^{L_2\text{-wait}}$, as opposed to simple policies like FIFO and NEAREST-NEIGHBOR.

5.3.1. Low load

As described in Section 5.2, we ran three simulation series. The results for the first series are shown in Fig. 2. The policy REPLAN provided the best results, whereas FIFO was completely inefficient. Considering the load histograms in Fig. 3, all policies seemed to work in an adequate way, with FIFO obviously being almost unstable.

For the next series, we used the "winner" policy C (REPLAN) as a competitor for the policies E and F. The results are shown in Fig. 4 . The best results were provided by policy F (GREEDY-LATE). The performance of policy E (GREEDY-DRIVE) is as bad as FIFO's. The load histograms in Fig. 5 indicate the stability of the policies.

Again, we used the "winner" policy F (GREEDY-LATE) of the last series as comparing element for the policies G–N. As shown in Fig. 6, reoptimization policies beat all the others by a substantial margin. There are no differences between the specific reoptimization policies, except the maximum waiting time (Fig. 7). For this objective, reoptimization

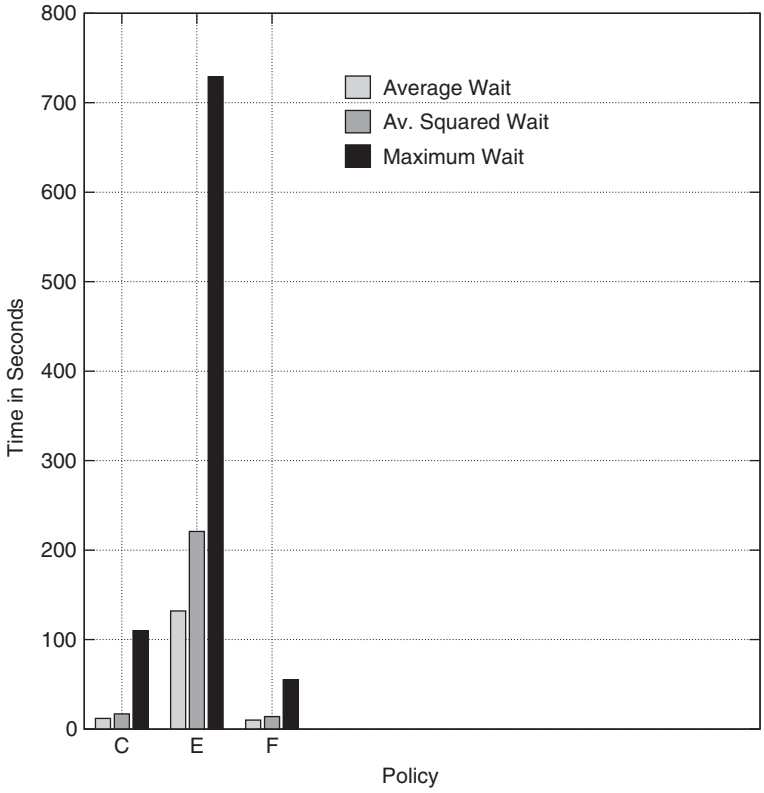

Fig. 4. Results of policies for low system load.

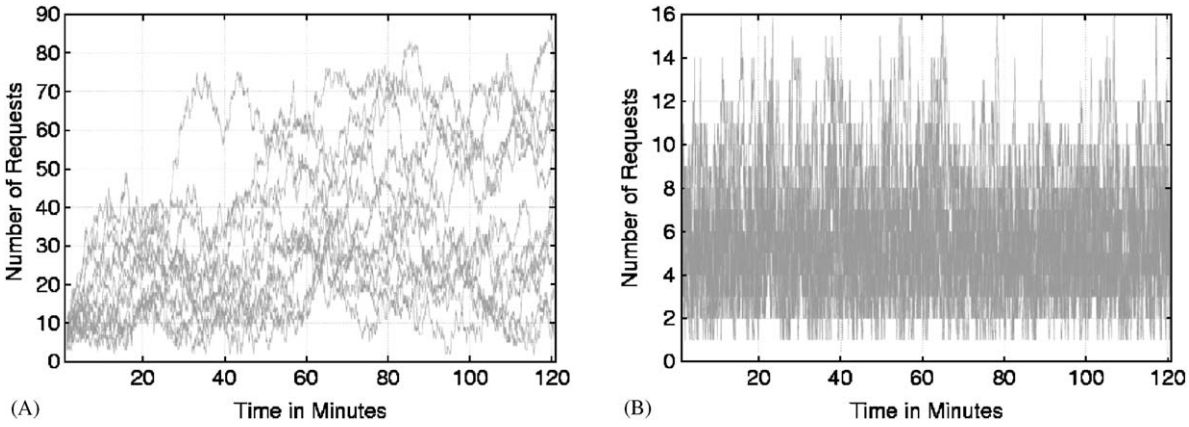

Fig. 5. Load histograms of policies at low system load: (A) policy E and (B) policy F.

policies w.r.t. $c^{L_2\text{-wait}}$ provide a far better performance than reoptimization policies w.r.t. c^{avgwait}. This is plausible, since the penalty for long waiting times is much more emphasized in the quadratic case than in the linear case. The load histograms are shown in Fig. 8.

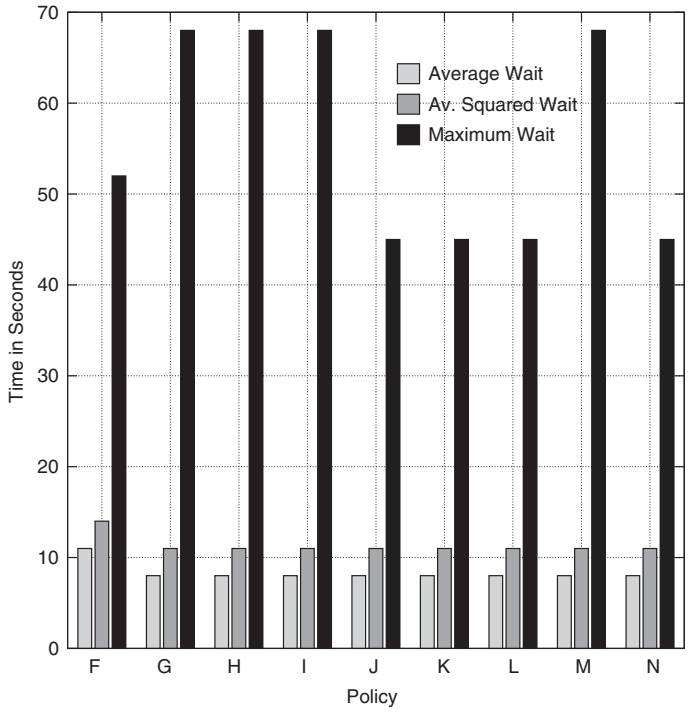

Fig. 6. Results of policies for low system load.

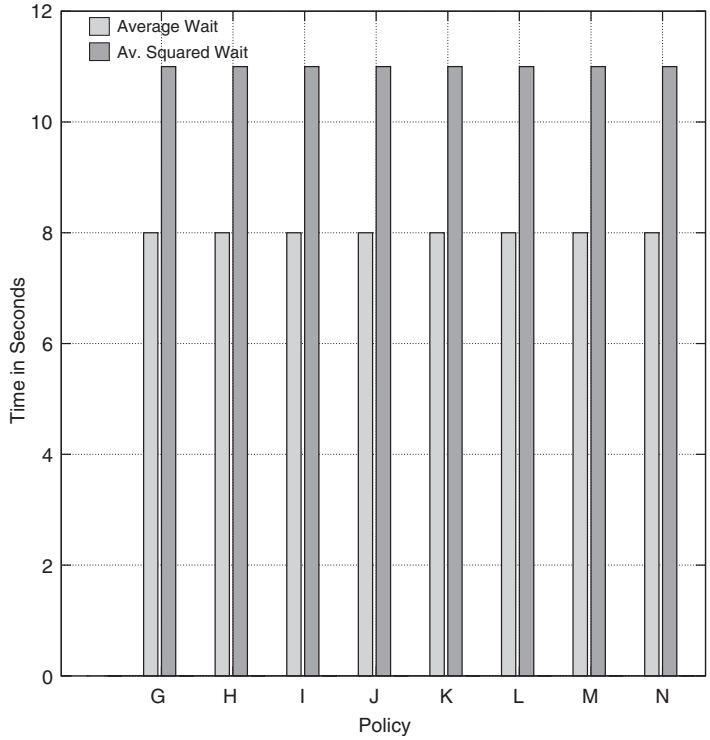

Fig. 7. Enlarged bar chart from Fig. 6 without maximal waiting times.

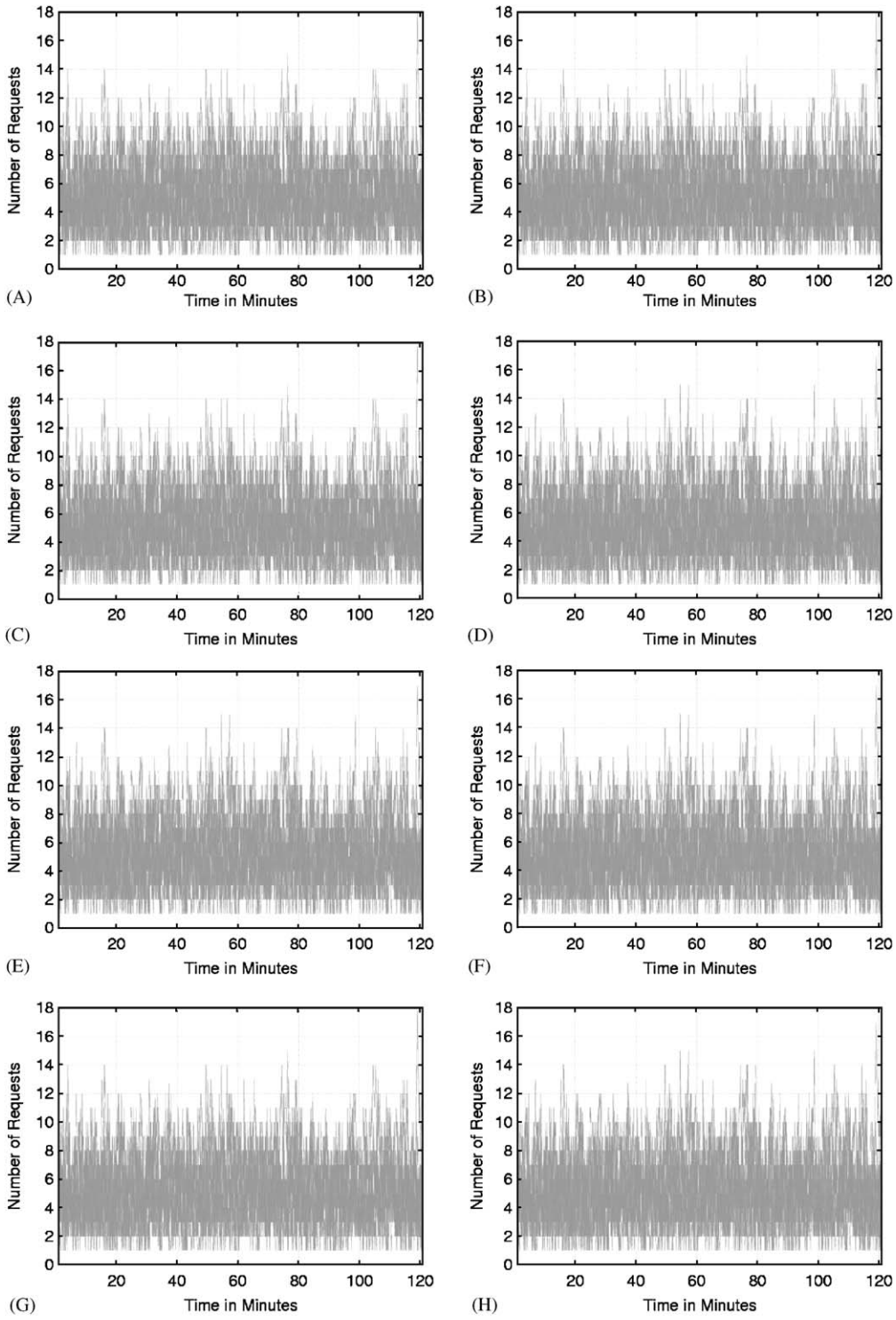

Fig. 8. Load histograms of policies at low system load: (A) policy G, (B) policy H, (C) policy I, (D) policy J, (E) policy K, (F) policy L, (G) policy M and (H) policy N.

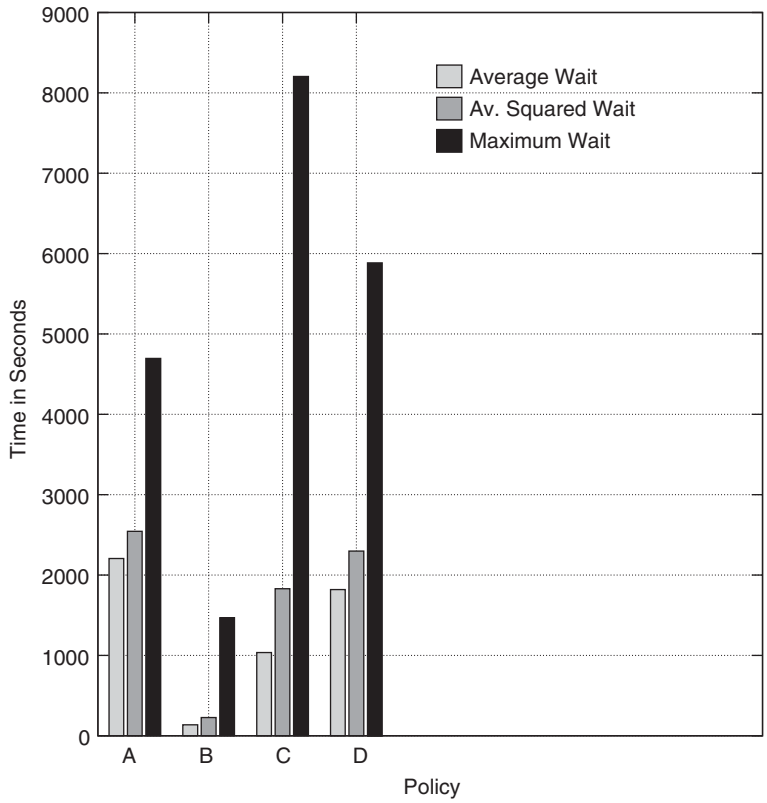

Fig. 9. Results of policies for high system load.

5.3.2. High load

In the following, we describe the results under high load. We expected an even better potential for the exact reoptimization policies, since at any time there are more known requests to plan with. The results for the first series are displayed in Fig. 9. We see here, and, even more so, in the load histograms in Fig. 10, that policies A, C, and D are extremely instable. The number of unserved known requests is constantly increasing. Policy B (NEAREST-NEIGHBOR), however, provided the best results in this series and seemed to achieve a stable performance.

Using policy B (NEAREST-NEIGHBOR) as the "winner" of the previous series, the results for the policies E and F are shown in Fig. 11. Like in the corresponding low load series, policy F (GREEDY-LATE) provided the best results, whereas policy E (GREEDY-DRIVE) is completely instable (see Fig. 12). This seemed very surprising, at first glance. A reduction of empty moves is very often considered synonymous to efficiency. In the multi-server case, however, a reduction of empty moves does not lead to a fast service. In principle it is conceivable that all but one server are idle if the only moving server can serve the requests essentially without empty moves.

The results of the third series, shown in Fig. 13, demonstrate the power of reoptimization policies (Fig. 14). All policies are (experimentally) stable in our sense (see Fig. 15).

Especially interesting is the following fact: the reoptimization policies w.r.t. $c^{L_2\text{-wait}}$ provide almost the same average waiting times as the policies reoptimizing w.r.t. c^{avgwait}; they achieve, however, by far better maximal waiting times. This means that essentially one can choose $c^{L_2\text{-wait}}$ as reoptimization goal without regret, no matter whether fairness is more or less important.

Policy REOPT-TWOOPT yields a surprisingly weak improvement over policy REOPT-BESTINSERT. It should be noted, however, that the reoptimization model only decides one single assignment in one reoptimization run, whereas previous assignments stay fixed. Remember that this is because the former requests have already entered their assigned elevators' local waiting queues. Because of this, REOPT-TWOOPT can simply not perform so many feasible exchanges.

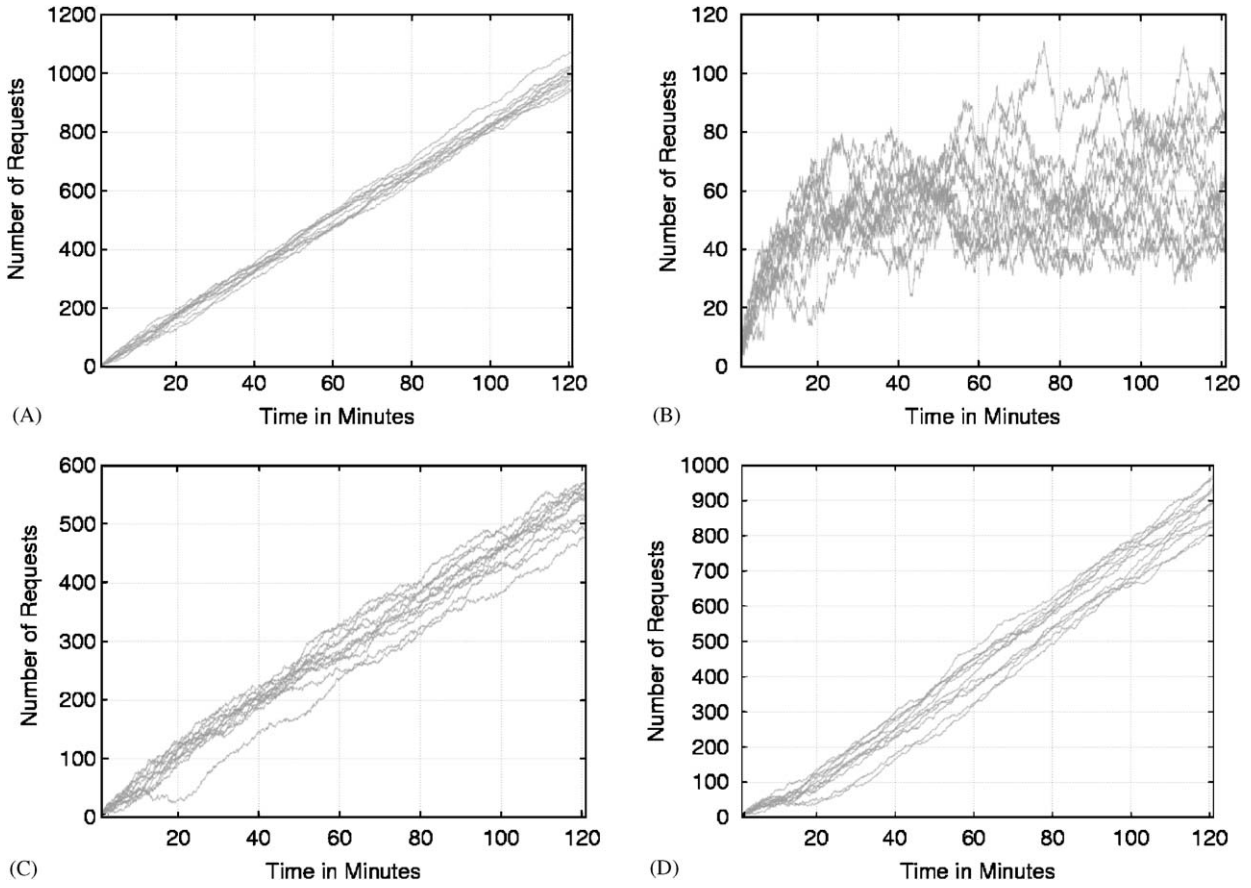

Fig. 10. Load histograms of policies at high system load: (A) policy A, (B) policy B, (C) policy C and (D) policy D.

The exact reoptimization methods provided almost 10% better performance than the heuristic reoptimization methods (see Fig. 14). We have to admit that this is rather an incremental improvement over heuristic reoptimization methods, in particular if compared to the improvement that the heuristic integrated reoptimization policies yield over the other policies. However, in view of the single free assignment decision per reoptimization, it is rather remarkable that, w.r.t. both c^{avgwait} and $c^{L_2\text{-wait}}$, policy REOPT-ZIBDIP, based on dynamic column generation, can still squeeze out almost 10% in terms of average waiting times against REOPT-BESTINSERT and REOPT-TWOOPT.

5.3.3. Capacity

To give a very compact view on what can be achieved with mathematical optimization for our idealized cargo elevator model, we compared FIFO (representing rule-based planning), NEAREST-NEIGHBOR (representing heuristically optimized planning), and REOPT-ZIBDIP w.r.t. $c^{L_2\text{-wait}}$ (representing mathematically optimized planning) in another experiment: we investigated how many elevator a policy needs, such that the *average flow time* does not exceed a certain time threshold. We used a threshold of 40 s for our systems (see Fig. 16(A)).

Subsequently, we used the number of elevators that the winner policy needed to reach the threshold also for the other two policies; we wanted to know how substantial the threshold would be violated (see Fig. 16(B)).

Qualitatively, the result is not surprising. What is surprising is the amount of improvement that can be achieved with mathematical optimization. That REOPT-ZIBDIP can save three out of eight elevators against FIFO and still one out

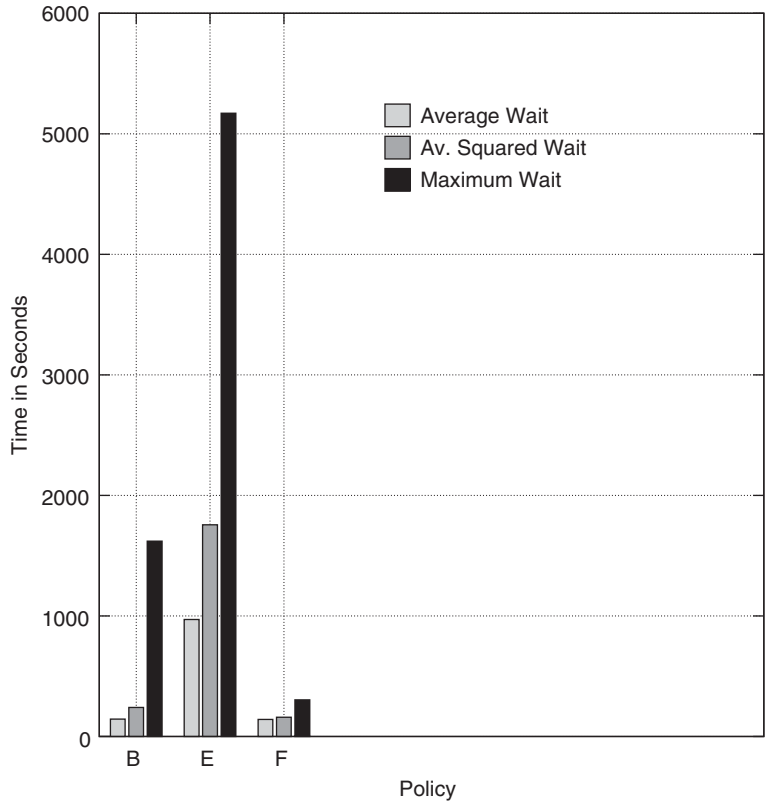

Fig. 11. Results of policies for high system load.

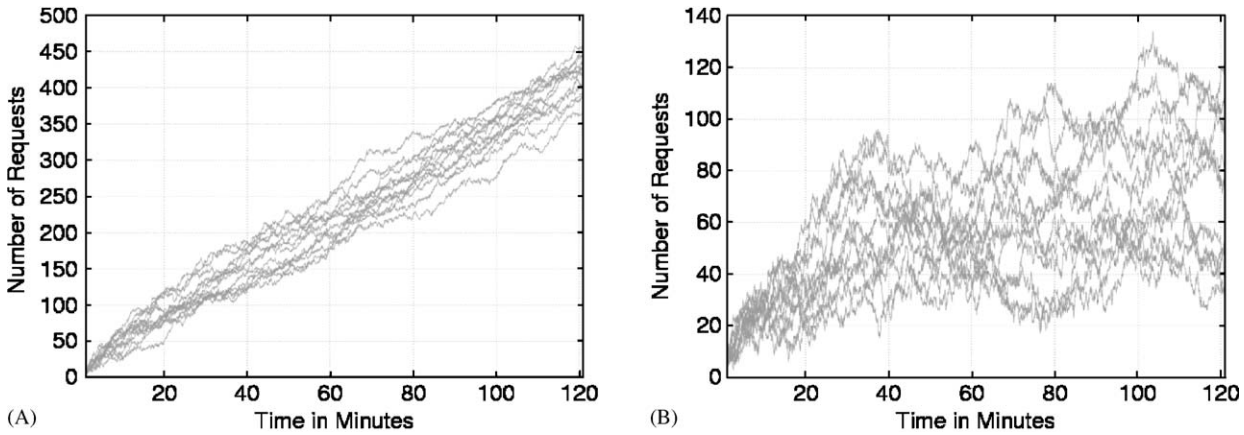

Fig. 12. Load histograms of policies at high system load: (A) policy E and (B) policy F.

of six elevators against NEAREST-NEIGHBOR is significant. Moreover, NEAREST-NEIGHBOR significantly degrades in performance when used with five elevators. Even more so: if FIFO has to get away with five elevators (no problem for REOPT-ZIBDIP), then the system breaks down completely (average waiting time of several thousands of seconds).

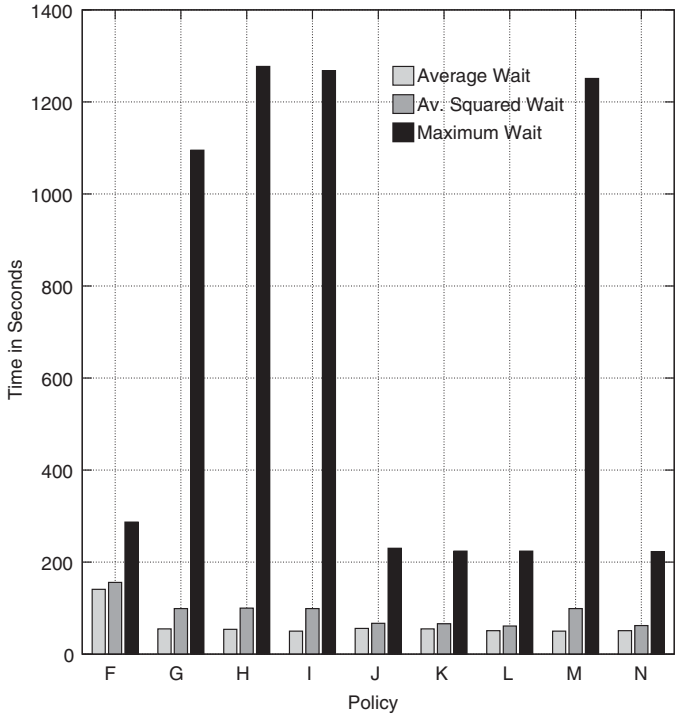

Fig. 13. Results of policies for high system load.

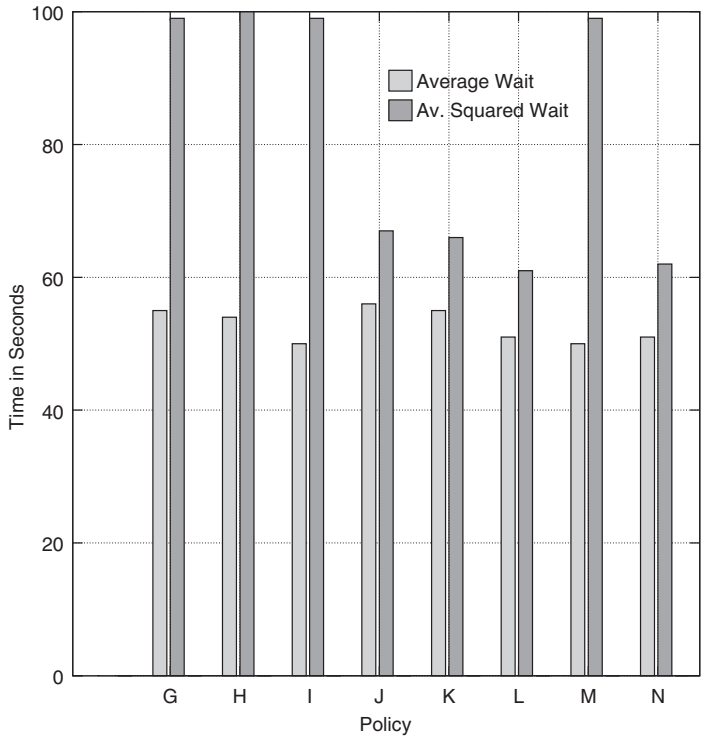

Fig. 14. Enlarged bar chart from Fig. 13 without maximal waiting times.

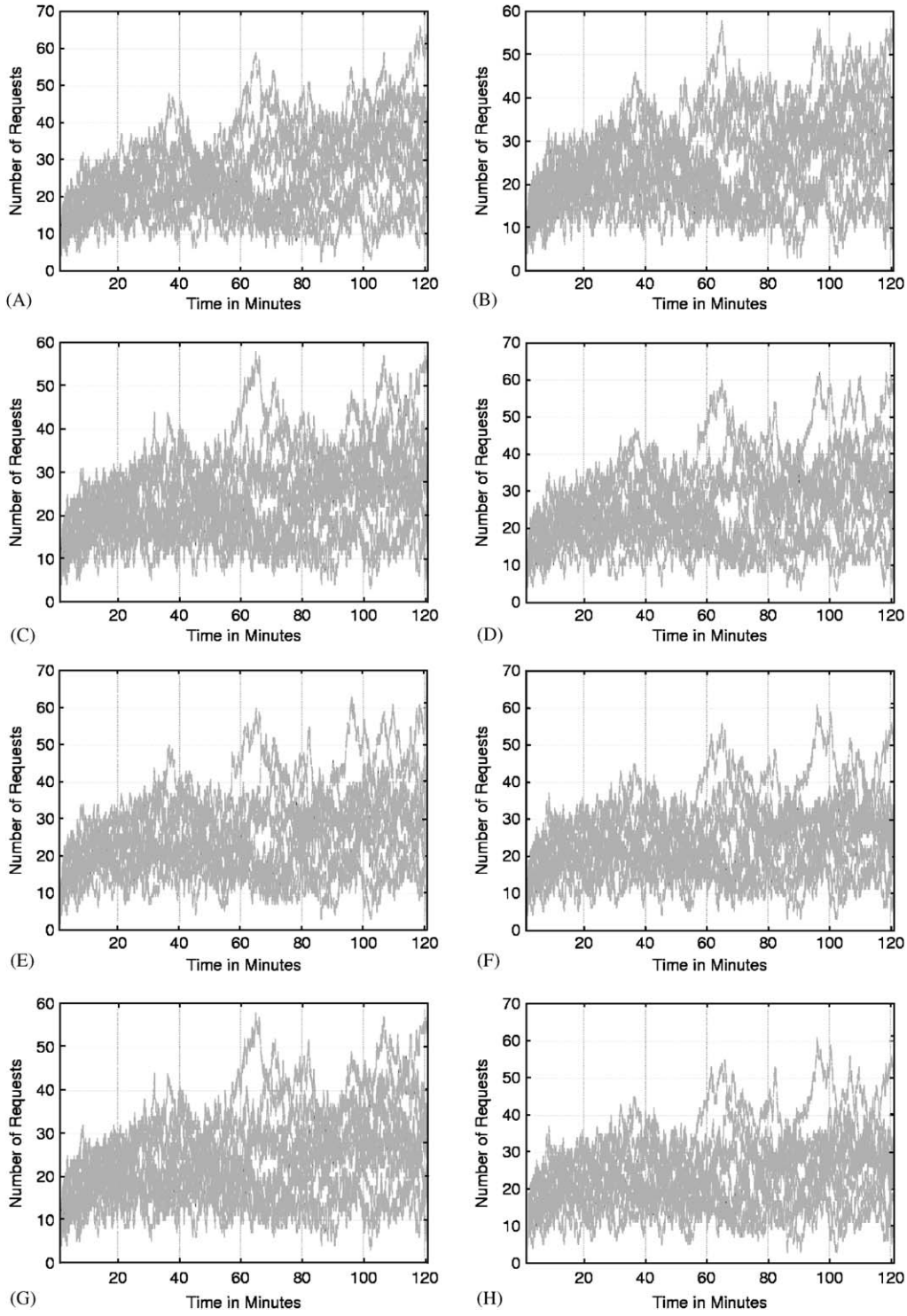

Fig. 15. Load histograms of policies at high system load: (A) policy G, (B) policy H, (C) policy I, (D) policy J, (E) policy K, (F) policy L, (G) policy M and (H) policy N.

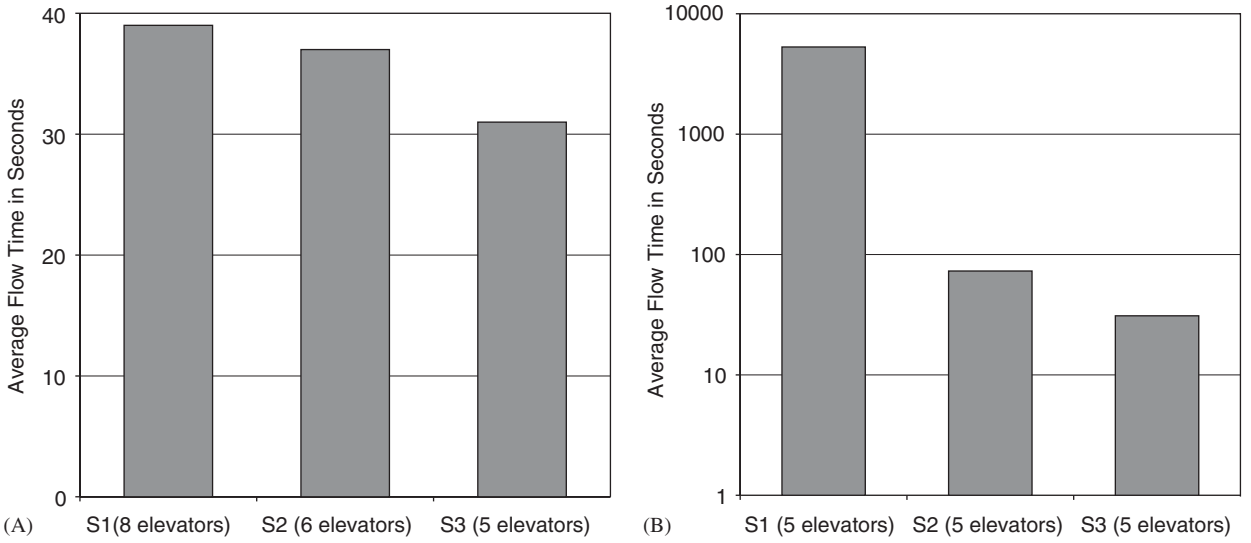

Fig. 16. Results for cargo elevator systems: (A) results (linear scale). Number of elevators for systems with 16 floors using the policy FIFO (S1), NEAREST-NEIGHBOR (S2) and REOPT-ZIBDIP (S3), such that the average flow time does not exceed the time threshold of 40 s and (B) results (semilog scale). Systems with five elevators operating on 16 floors using the policy FIFO (S1), NEAREST-NEIGHBOR (S2) and REOPT-ZIBDIP (S3).

6. Extensions

We already mentioned that the reoptimization objective can be chosen to be more complicated than just the sum of squared waiting times. In practice, it has proven to be useful to introduce a low-weight penalty for the driving distance. This acts as a perturbation of the reoptimization problem and is therefore technically desirable.

We also mentioned that the degrees of freedom for each reoptimization are quite restricted because former assignments are irrevocable. We have experimented with a delayed assignment model: each assignment is only tentative and can be revised, i.e., the request stays in the global waiting queue. This remains so until an elevator needs to get access to a special request in the middle of the global waiting queue. At that point in time, all requests waiting in front of the special request have to flow into the local queues of their currently assigned elevators. With this method, an additional improvement, though not too large, can be achieved. This delayed assignment, however, modifies the basic rules of the game, and all new heuristics might be possible that also show better performance. Therefore, we did not present any of these results in this work.

Lower bounds for a posteriori optimal long-term dispatches can be computed in principle. We found out, however, that the knowledge about the future requests is so strong that even this so-called experimental competitive analysis gives no relevant information about by how much our policies could still be improved. The implementation of a so-called fair or non-abusive adversary is still in progress and subject of future research work.

7. Conclusion

In dynamic multi-elevator dispatching with the goal of small average and maximal waiting times, reoptimization w.r.t. the sum of the squared waiting times is the best policy we could find. It outperforms common heuristic policies by a surprisingly large margin.

Compared with the results about reoptimization w.r.t. the average waiting time, it is apparent that the change in the reoptimization objective can keep the maximal individual waiting time low without significantly increasing the average waiting time. In our view, a proper adaption of static methods in the spirit of [6] is, therefore, to use the quadratic waiting time objective.

It would be desirable to compare reinforcement learning and evolutionary policies against our reoptimization policies. Since the parameter tuning of such policies is rather an art than a science (and we are not particularly skilled in

implementing these special methods properly), we invite the specialists in that area to arrange for a shoot-out of policies.

The capacity planning of transport systems like the elevator system in the distribution center at Herlitz (how many elevators do we need?) is very often done w.r.t. specific control policies, mostly FIFO. In this context, it is interesting to see that one can save a substantial amount of elevators at the same level of average waiting time if capacity is planned based on the use of our reoptimization policy. To what extent this observation can be implemented into the real system is the subject of future research.

References

[1] N. Ascheuer, Hamiltonian path problems in the on-line optimization of flexible manufacturing systems, Dissertation, Technische Universität Berlin, 1995, URL ⟨ftp://ftp.zib.de/pub/zib-publications/reports/TR-96-03.ps.Z⟩.

[2] N. Ascheuer, Amsel—a modelling and simulation environment library, developed at Konrad-Zuse-Zentrum für Informationstechnik, Berlin, Available online: ⟨http://www.zib.de/Optimization/Software/Amsel⟩.

[3] N. Ascheuer, M. Grötschel, S.O. Krumke, J. Rambau, Combinatorial online optimization, in: P. Kall, H.-J. Lüthi (Eds.), Proceedings of the International Conference on Operations Research (OR'98), Gesellschaft für Operations Research e.V. (GOR), Springer, Berlin, 1998, pp. 21–37, URL ⟨http://www.zib.de/PaperWeb/abstracts/SC-98-24/⟩.

[4] N. Ascheuer, S.O. Krumke, J. Rambau, Online dial-a-ride problems: minimizing the completion time, in: Proceedings of the 17th International Symposium on Theoretical Aspects of Computer Science, vol. 1770, Springer, Berlin, 2000, pp. 639–650, URL ⟨http://www.zib.de/PaperWeb/abstracts/SC-98-34/⟩.

[5] R.W. Bent, P. van Hentenryck, Scenario-based planning for partially dynamic vehicle routing with stochastic demands, Brown University, 2002, preprint.

[6] D. Bertsimas, D. Simchi-Levi, A new generation of vehicle routing research: robust algorithms, addressing uncertainty, Oper. Res. 44 (1996) 286–304.

[7] D.J. Bertsimas, A mathematical programming approach to stochastic and dynamic optimization problems, Working Paper 3668-94, Sloan School of Management, MIT, 1994.

[8] D.J. Bertsimas, G. van Ryzin, A stochastic and dynamic vehicle routing problem in the euclidean plane, Oper. Res. 39 (1991) 601–615.

[9] D.J. Bertsimas, G. van Ryzin, Stochastic and dynamic vehicle routing in the euclidean plane with multiple capacitated vehicles, Oper. Res. 41 (1993) 60–76.

[10] A. Coja-Oghlan, S.O. Krumke, T. Nierhoff, A heuristic for the stacker crane problem on trees which is almost surely exact, in: T.Ibaraki, N.Katoh, H.Ono (Eds.), Algorithms and Computation, Lecture Notes in Computer Science, vol. 2906/2003, Springer, Berlin, 2003, pp.605–614.

[11] R.H. Crites, A.G. Barto, Improving elevator performance using reinforcement learning, in: S. Touretsky, D.C. Mozer, M.E. Hasselmo (Eds.), Advances in Neural Information Processing Systems, vol. 8, MIT Press, Cambridge, MA, 1996.

[12] R.H. Crites, A.G. Barto, Elevator group control using multiple reinforcement learning agents, Mach. Learning 33 (2–3) (1998) 235–262.

[13] E. Feuerstein, L. Stougie, On-line single-server dial-a-ride problems, Theoret. Comput. Sci. 268 (2001) 91–105.

[14] P. Friese, Echtzeitsteuerung von Multi-Server-Transportsystemen mit Reoptimierungsalgorithmen, Diplomarbeit, Technische Universität, Berlin, 2003.

[15] M. Gendreau, F. Guertin, J.-Y. Potvin, É. Taillard, Parallel tabu search for real-time vehicle routing and dispatching, Transportation Sci. 33 (4) (1999) 381–390.

[16] M. Gendreau, J.-Y. Potvin, Dynamic vehicle routing and dispatching, in: T.G. Crainic, G. Laporte (Eds.), Fleet Management and Logistics, Kluwer Academic Publishers, London, 1998, pp. 115–126.

[17] GNU Compiler Collection, Software under the GNU Public Licence (GPL), Available online: ⟨http://www.gnu.org/gcc⟩.

[18] M. Grötschel, D. Hauptmeier, S.O. Krumke, J. Rambau, Simulation studies for the online dial-a-ride-problem, Konrad-Zuse-Zentrum für Informationstechnik Berlin, Extended abstract accepted for presentation at Odysseus 2000, First Workshop on Freight Transportation and Logistics, Crete, 2000 (1999), preprint SC 99-09, URL ⟨http://www.zib.de/PaperWeb/abstracts/SC-99-09/⟩.

[19] M. Grötschel, S.O. Krumke, J. Rambau, L.M. Torres, Online-dispatching of automobile service units, in: U. Leopold-Wildburger, F. Rendl, G. Wäscher (Eds.), Operations Research Proceedings, Springer, Berlin, 2002, pp. 168–173, URL ⟨http://www.zib.de/PaperWeb/abstracts/ZR-02-44/⟩.

[20] M. Grötschel, S.O. Krumke, J. Rambau, T. Winter, U.T. Zimmermann, Combinatorial online optimization in real time, in: M. Grötschel, S.O. Krumke, J. Rambau (Eds.), Online Optimization of Large Scale Systems, Springer, Berlin, 2001, pp. 679–704, URL ⟨http://www.zib.de/PaperWeb/abstracts/ZR-01-16/⟩.

[21] D. Hauptmeier, S.O. Krumke, J. Rambau, The online dial-a-ride problem under reasonable load, in: Proceedings of the Fourth Italian Conference on Algorithms and Complexity, Lecture Notes in Computer Science, vol. 1767, Springer, Berlin, 2000, pp. 137–149, URL ⟨http://www.zib.de/PaperWeb/abstracts/SC-99-08/⟩.

[22] D. Hauptmeier, S.O. Krumke, J. Rambau, H.C. Wirth, Euler is standing in line—dial-a-ride problems with FIFO-precedence-constraints, Discrete Appl. Math. 113 (2001) 87–107 URL ⟨http://www.zib.de/PaperWeb/abstracts/SC-99-06/⟩.

[23] S. Ichoua, M. Gendreau, J.-Y. Potvin, Diversion issues in real-time vehicle dispatching, Transportation Sci. 34 (4) (2000) 426–438.

[24] ILOG-CPLEX-9.0, Mathematical Programming Optimizer Software, Information available online: ⟨http://www.ilog.com/products/cplex⟩.

[25] H. Kellerer, T. Tautenhahn, G. Woeginger, Approximability and nonapproximability results for minimizing total flow time on a single machine, in: Proceedings of the Symposium on the Theory of Computing, 1996.

[26] J. Koehler, D. Ottiger, An AI-based approach to destination control in elevators, AI Magazine 23 (2002) 59–78.
[27] J. Koehler, K. Schuster, Elevator control as a planning problem, in: S. Chien, S. Kambhampati, C.A. Knoblock (Eds.), Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems, AAAI, 2000, pp. 331–338.
[28] S.O. Krumke, W.E. de Paepe, D. Poensgen, L. Stougie, News from the online traveling repairman, Theoret. Comput. Sci. 295 (2003) 279–294.
[29] S.O. Krumke, J. Rambau, L.M. Torres, Realtime-dispatching of guided and unguided automobile service units with soft time windows, in: R.H. Möhring, R. Raman (Eds.), Algorithms—ESA 2002, 10th Annual European Symposium, Rome, Italy, September 17–21, 2002, Proceedings, Lecture Notes in Computer Science, vol. 2461, Springer, Berlin, 2002, URL ⟨http://www.zib.de/PaperWeb/abstracts/ZR-01-22⟩.
[30] A.S. Minkoff, A Markov decision model and decomposition heuristic for dynamic vehicle dispatching, Oper. Res. 41 (1993) 77–90.
[31] M. Müller, Online-optimierung und Layout-Planung von Aufzugsystemen, Master's Thesis, Technische Universität, Berlin, 2000.
[32] W.B. Powell, T.A. Carvalho, G.A. Godfrey, H.P. Simão, Dynamic fleet management as a logistic queueing network, Ann. Oper. Res. 61 (1995) 165–188.
[33] H.N. Psaraftis, Dynamic vehicle routing: status and prospects, Ann. Oper. Res. 61 (1995) 143–164.
[34] H.-J. Siegert, Simulation zeitdiskreter Systeme, Oldenbourg, München, Wien, 1991.
[35] A. Stagno, P. Chénais, T.M. Liebling, QOBJ modeling, OR Spektrum (20) (1998) 109–122.
[36] K.Q. Zhu, K.-L. Ong, A reactive method for real time dynamic vehicle routing problem, in: Proceedings of the 12th ICTAI, 2000.